Contents

Chapter 1

Introduction

1.1 Design Methodolgy

There are two design methodolgies.

Embedded buncha things.

Integrated Circuits the sexy stuff that we would love talk about them.

There three main type of ICs.

Programmable Logic Devic buncha AND and OR.

Field Programmable Gate Arrays buncha LUT. they are relatively cheap and easy to produce. programmed easily, usually done with USB - can't get easier! can it?!.

Application Specific Integrate Circuite customized but non-programmable. Longer design time and higher cost but high preformance and lower power consumptions. Good for big boys that are done playing with their FPGA. but first they gotta send their design (*Fabrication patten*) to the factory (*FAB*).

Now you may ask why should I use programmable logic. Firstly, because it is cool and eviroment friendly. DON'T be a climate change denier baster please :)). Here are more reason

1. Reduce time to mark (TTM).
2. used for prototyping.
3. Reconfigurable computing
4. Custom computing
5. Reusability for different designs (Cant get greener than this :p).

1.2 Hardware Description Languages

describes hardware, it is in the name -_-!

1. Popular HDLs (IEEE standard)

   (a) Verilog (we gonna use this mostly because it's the cooler kid)
   (b) VHDL (used for modeling mostly and thus not all constructs are synthesizable)

2. other HDLs (not standard! not good! so no need to know baby)

HDLs are concurrent and not sequential - just like the real world. It also has timing that is you can have clocks for sequential circuits. furhtermore, it supports desing hierarchy (donnu what this is).

$$\text{Logic Synthesis} = \text{Translation} \rightarrow \text{Optimization} \rightarrow \text{Mapping}$$

We use ASM, *Algorithm State Machine* for large scale integration. and we use CAD/IDE (HDLs) for very large scale integrattion.

Definition (Netlist): HDL describing logic gates.

1.3 Design flow

1. Design specification- logical and physical.
2. Behavioral description- for the circuit.
3. RLT description via HDLs.
4. Functional verification and testing. if you suck go back to item 3.
5. Logic Synthesis.
6. Gate-level netlist. produced by logic synthesis.
7. Logical verification and testing. use the gate list and run test. if it sucks go back to item 3.
8. Floor planning, automatic place & route. usually done automaticly in FPGA and manually in ASIC. this step is called back-end design or physical design.
9. Physical layout - done with CAD and sent to FAB. GDStool
10. Layout verification. before you send your stupid design check it. done by barghis and not us.
11. Implementation. enjoy your shitty devic now.

Chapter 2

ASM Chart

The **algorithmic state machine** is a method for designing finite state machines which describe the sequential operations of a digital system. Its a behavioral model using flowcharts suitable for LSI.

There are three main elements to a ASM chart. State box, conditional box and a decision box.

in state and conditional boxes, the commands takes one clock cycle to be executed. conditional box is combinational. decision box contains binary expressions. **ASM block** is composed of one state box and all the decision and conditional boxes connected to the exit path of the state box. It represents what happens in the system during one clock cycle.

Chapter 3

Verilog

3.1 Behavioral and structural design

Behavioral design describes circuit as an algorithm and structural design describes explicit circuit elements. A behavioral example for a full adder.

```
module adder(a,b,cin,s,cout);
    output s,cout;
    input a,b,cin;
    reg s, cout;
    always @(a or b or cin)
    begin
        s = a^b^cin;
        cout = a&b | a& cin | b& cin;
    end
endmodule
```

And an example for a structural design of the same full adder.

```
module adder(a,b,cin,s,cout)
    output s,cout;
    input a,b,cin;

    wire w1,w2,w3,w4,w5;
    xor g1(w1,a,b);
    xor g2(s,w1,cin);
    and g3(w2,a,b);
    and g4(w3,a,cin);
    and g5(w4,b,cin);
    or g6(w5,w2,w3);
    or g7(cout,w4,w5);
endmodule
```

Number are specified by:

- Radix ('b,'h,'o,'d)
- Bit-length
- Value

for example

- 4'b1011

- 12'habc
- 16'd255

3.2 4-value logic

The set of value are $\{0, 1, x, z\}$. x is for unknown and z is for high impedence. Signal strengths are (in order):

1. supply (Driving)
2. strong (Driving)
3. pull (Driving)
4. large (Storage)
5. weak (Driving)
6. medium (Storage)
7. small (Storage)
8. highz (High impedence)

wire is used to represent connections between hardware elements (default = z). reg for hardware as well but it retains its value until next assignment (default = x).

3.3 Wire and Reg

1. Syntax

```
    wire/reg [msb_index : lsb_index] data_id;
```

2. Example

```
    wire a;
    wire [7:0] bus;
    wire [31:0] busA, busB, busC;
    reg clock;
    reg [0:40] virtual_addr;
```

modules are models of hardware. internals not visible to enviroment. internals can be changed as long as the interface (ports) is not changed.

```
    module fulladd4 (sum, c_out,a,b,c_in)
    ...
    endmodule
```

Ports are the terminals (pins) which have three types (in, out, inout). For port declaration

```
    input a,b,sel;
    input signed [15:0] a,b;
    output signed [31:9] result;
    output reg signed [32:1] sun;
    inout [15:12] addr;
```

In a module, inputs are always of type net but output can be reg as well. inout can be of type net only. We can implement delays as such

3.4 Delay

```
    and #(delay-time) a1 (out,i1,i2);
    and #(rise-val,fall-val) a1 (out,i1,i2);
    and #(min:typ:max,min:typ:max) a1 (out,i1,i2);
    bufif0 #(rise-val,fall-val,turnoff-val) a1 (out,in,control);
```

3.5 Data types

3.5.1 Integer

signed numbers.

```
    integer [7:0] tmp;
```

3.5.2 Real

default value is zero

```
    real delta;
    delta=4e10;
    delta=2.13;
```

3.5.3 Time

at least 64 bit and *$time* is a system function that gives current simulation time.

```
    time save_sim_time;
    initial
        save_sim_time = $time;
```

3.6 Vector and array

3.6.1 Vectors

```
    wire/reg [msb_index : lsb_index] data_id;

    wire a; //single bit
    wire [7:0] bus; //8-bit vector
    wire [31:0] busA, busB, busC;
    reg clock;
    reg [0:40] virtual_addr;
```

3.6.2 Arryas

only one-dimensional and allowed for **reg,integer,time**

```
    <data_type> <var_name> [start_idx : end_idx];

    integer count[0:7];
    reg bool[31:0];
```

```
    time chk_point[1:100];
    reg [4:0] port_id[0:7];
    integer matrix[4:0][4:0]; // illegal
```

3.7 Assignments

3.7.1 Continuous assignments

assigns values to nets, happens whenever simulation causes the value of the right-hand side to change. used for combinational logic. can happen outside of any blocks and cant be used in **always** or **initial**

```
    wire [15:0] sum, a, b; // declaration of 16-bit vector nets
    wire cin, cout; // declaration of 1-bit (scalar) nets
    assign {cout, sum} = a + b + cin;
```

3.7.2 Procedural Assignments

drive value to **reg**. holds value of the assignment until the next assignment. occur in blocks. In **initial** block it is only executed once, but in **always** is repeated continuously. for combinational and sequential.

```
    reg Clock, Enable, Load, Reset;
    reg [7:0] Data;
    parameter HalfPeriod = 5;
    initial
        begin
            Clock = 0;
            #(HalfPeriod) Clock = ~Clock;
        end
```

3.8 Delay

3.8.1 Inertial Delay

Like the delay induced by combinational elements. delayed evaluation

```
module testbench;
    reg a,b;
    always @(a)
    begin
        $display($time);
        #10 b=~a;
    end
    initial
    begin
        a=1'b0;
        #15 a=1'b1;
        #3 a=1'b0;
        #15 a=1'b1;
        #11 a=1'b0;
    end
endmodule
```

3.8.2 Transport Delay

Delayed assignment

```
module testbench;
    reg a,b,c;
    always @(a)
    begin
        $display($time);
        b <= #10 ~a;
    end
    initial
    begin
        a=1'b0;
        #15 a=1'b1;
        #3 a=1'b0;
        #15 a=1'b1;
        #11 a=1'b0;
    end
endmodule
```

In combinational if

Definition:

No dalay use blocking

Interial delay use blocking

Transport delay use non-blocking

In sequential logic if

Definition:

No dalay use non-blocking

with delay use non-blocking and transport delay.

3.9 More keywords

3.9.1 wait

wait for a certain condition to be true and is level-sensitive.

```
always
    wait (CountEnable) #20 count = count+1;
```

3.9.2 case

```
reg [1:0] alu_control;
case (alu_control)
    2'd0 : y = x + z;
    2'd1 : y = x - z;
    2'd2 : y = x * z;
    default : $display("Invalid ALU control signal");
endcase
```

```
module mux4-to-1 (out, i0, i1, i2, i3, s1, s0);
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out;
    always @(s1 or s0 or i0 or i1 or i2 or i3)
        case ({s1, s0})
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
        //default is selected when s1(or s2)=x or z
    endcase
endmodule
```

casez treats all z values as don't cares. casex treats all z and x values as don't cares.

```
module test;
    reg s1, s0;
    reg out;
    initial
    begin
        s1 = 'b0;
        s0 = 'bx;
        casex ({s1, s0})
        2'b01 : $display("01");
        2'bx1 : $display("x1");
        2'b0x : $display("0x");
        //this alternative is executed
        2'b00 : $display("00");
        default: $display("Default");
        endcase
    end
endmodule
```

3.9.3 repeat and forever

used only for modeling

```
initial
begin
    count = 0;
    repeat (128)
    begin
    $display("Count = %d", count);
    count = count + 1;
    // count: from 0 to 127
    end
end
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock; //best clock practice
en
```

3.9.4 function

no timing control and returns a single value and has at least an input. can be called in behavioral and structural models.

```
module parity;
    reg [31:0] addr;
    reg parity;
    always @(addr)
    begin
        parity = calcparity(addr);
        $display("Parity calculated = %b", calcparity(addr));
    end
    function calcparity;
        input [31:0] address;
        begin
            calcparity = ^address;
        end
    endfunction
endmodule
```

Example 3.1. A recursive example which needs the keyword automatic.

```
module function_auto ();
    function automatic [7:0] factorial;
        input [7:0] i_Num;
        begin
            if (i_Num == 1)
            factorial = 1;
            else
            factorial = i_Num * factorial(i_Num-1);
        end
    endfunction
    initial
    begin
        $display("Factorial of 1 = %d", factorial(1));
        $display("Factorial of 2 = %d", factorial(2));
        $display("Factorial of 3 = %d", factorial(3));
        $display("Factorial of 4 = %d", factorial(4));
        $display("Factorial of 5 = %d", factorial(5));
    end
endmodule

```

3.9.5 task

must be used if the procdure has any timing control, zero or more than one output argument. must be called within behavioral bodies.

```
module operation;
    reg [15:0] A, B;
    reg [15:0] AB_AND, AB_OR, AB_XOR;
    always @(A or B)
    begin
        bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
        //Argument passing by name not yet supported
    end
    //define task bitwise_oper
```

```
    task bitwise_oper;
    output [15:0] ab_and, ab_or, ab_xor;
    input [15:0] a, b;
        begin
        #10 ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
        end
    endtask
endmodule
```

3.10 Generate Statement

3.10.1 Generate Conditional Statement

synthesize different hardware based on conditions that resolve in compile. only one hardware gets synthesized.

```
//This module implements a parametrized multiplier
module multiplier (product, a0, a1);
    // Parameter Declaration. This can be redefined
    parameter a0_width = 8; // 8-bit bus by default
    parameter a1_width = 8; // 8-bit bus by default
    // Local Parameter declaration.
    // This parameter cannot be modified with defparam or
    // with module instance # statement.
    localparam product_width = a0_width + a1_width;
    // Port declarations
    output [product_width -1:0] product;
    input [a0_width-1:0] a0;
    input [a1_width-1:0] a1;
    // Instantiate the type of multiplier conditionally.
    // Depending on the value of the a0_width and a1_width
    // parameters at the time of instantiation, the appropriate
    // multiplier will be instantiated.
    generate
        if (a0_width <8) || (a1_width < 8)
            cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
        else
            tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
    endgenerate //end of the generate block
endmodule
```

3.10.2 Generate a unit multiple times